Dizajn sekvencijalnih kola

Sekvencijalno kolo je kolo koje ima interno stanje ili memoriju. Sinhrona sekvencijalna kola su kola kod kojih sve elemente kontroliše globalni signal za sinhronizaciju. Izlaz sekvencijalnih kola zavisi od stanja na ulazu, ali i od internog stanja. Interno stanje pamti uticaj prethodnih ulaznih signala. Dakle, izlaz sekvencijalnog kola zavisi od trenutnog, ali i prethodnog stanja na ulazu (ili čitave sekvence vrijednosti na ulazu kola). Upravo iz opisanog razloga, ova kola se označavaju kao sekvencijalna kola.

Osnovni memorijski elementi

c	q*
0	q
1	d

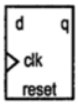
(a) D latch

clk	q*
0	q
1	q
↓	d

(b) positive-edge-triggered D FF

clk	q*
0	q
1	q
↓	d

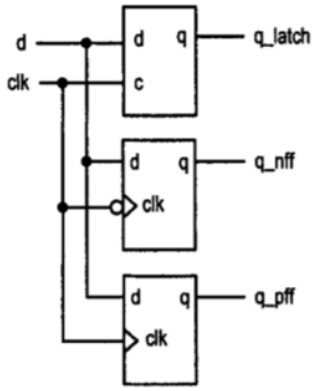
(c) negative-edge-triggered D FF

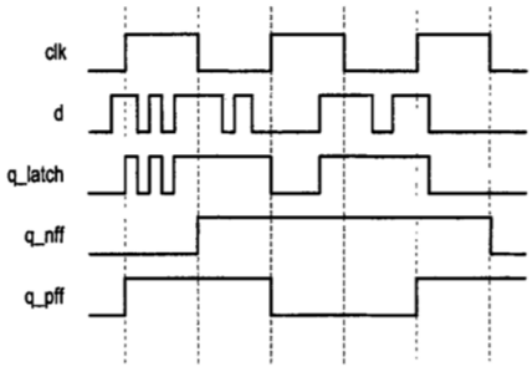


reset	clk	q*
1	-	0
0	0	q
0	1	q
0	↓	d

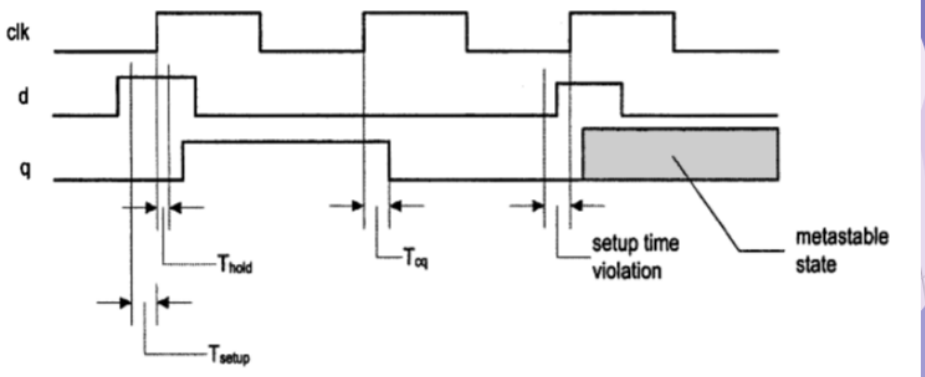
(d) D FF with asynchronous reset

Osnovni memorijski elementi



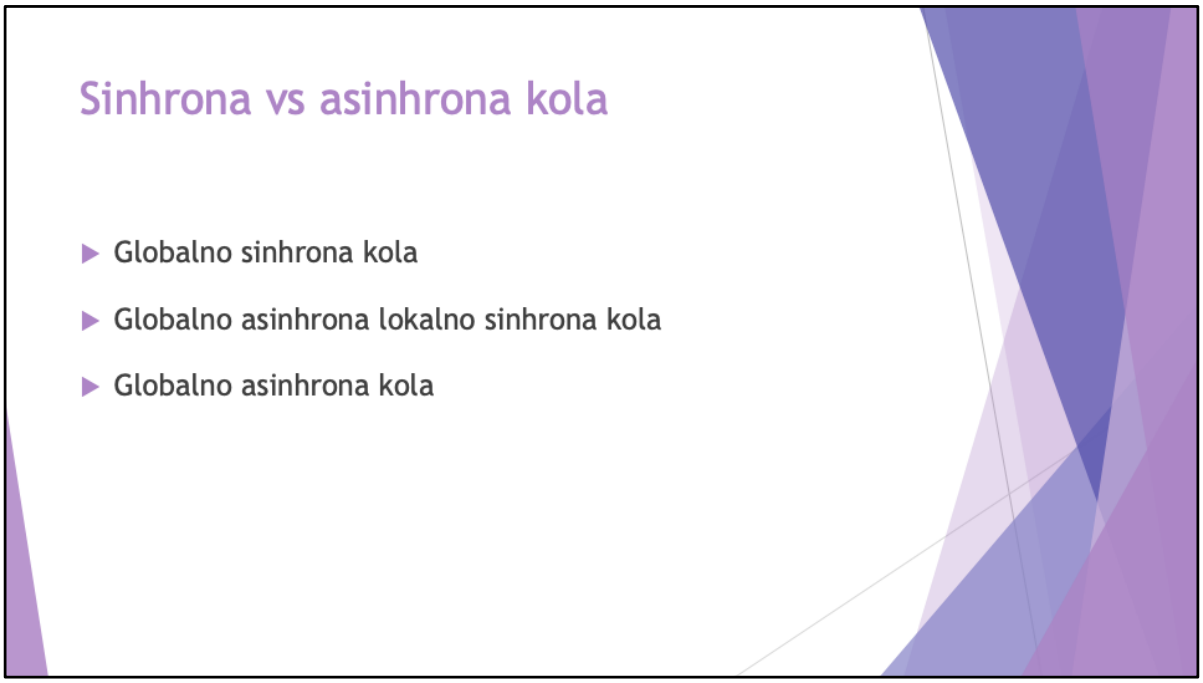


Osnovni memorijski elementi



Tcq (clock-to-q) je propagaciono kašnjenje potrebno da se signal d proslijedi na izlaz q nakon odgovarajuće ivice takta.
Tsetup (setup time) je vrijeme koje je potrebno da signal d bude stabilan prije odgovarajuće ivice takta.
Thold (hold time) je vrijeme koje je potrebno da signal d bude stabilan nakon odgovarajuće ivice takta.

Tcq grubo odgovara propagacionom kašnjenju kombinacionih kola. Tsetup i Thold su vremenska ograničenja kola. Oni specificiraju da d signal mora biti stabilan određeno vrijeme prije i poslije odgovarajuće ivice takta. Ukoliko se d signal promijeni u toku tog vremena (setup/hold time violation), DFF se može naći u metasatbilnom stanju gdje je vrijednost q neodređena.

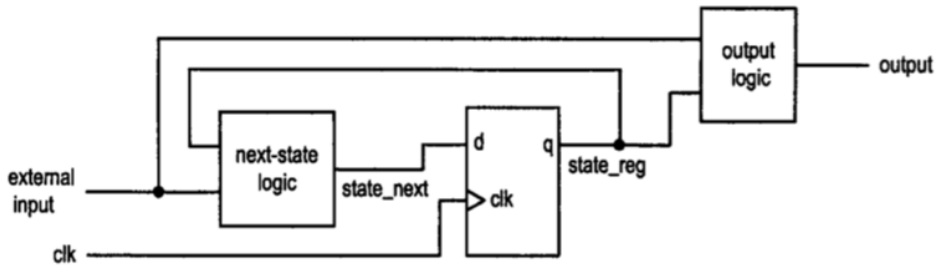


Globalno sinhrona kola koriste FF memorijske elemente, i svi FF-ovi su kontrolisani jednim takt signalom(clock). Sinhroni dizajn je najvažnija metodologija koja se koristi prilikom dizajniranja kompleksnih sistema. Pojednostavljuje sintezu, ali i verifikaciju i testiranje.

U pojedinim situcaijama, kada postoje izvjesna prostorna ograničenja, nije praktično sprovoditi isti takt do svih djelova sistema. Sistem se projektuje kao skup više podsistema, od kojih svaki ima svoj takt. Ukupno gledano, takav sistem je asinhron i potrebno je specifično interfejsno kolo među sistemima kako bi sistem pozdano funkcionisao.

Globalno asinhrona kola ne koriste takt signal. Stanje memorijskog elementa se mijenja nezavisno.

Osnovni model sinhronih kola

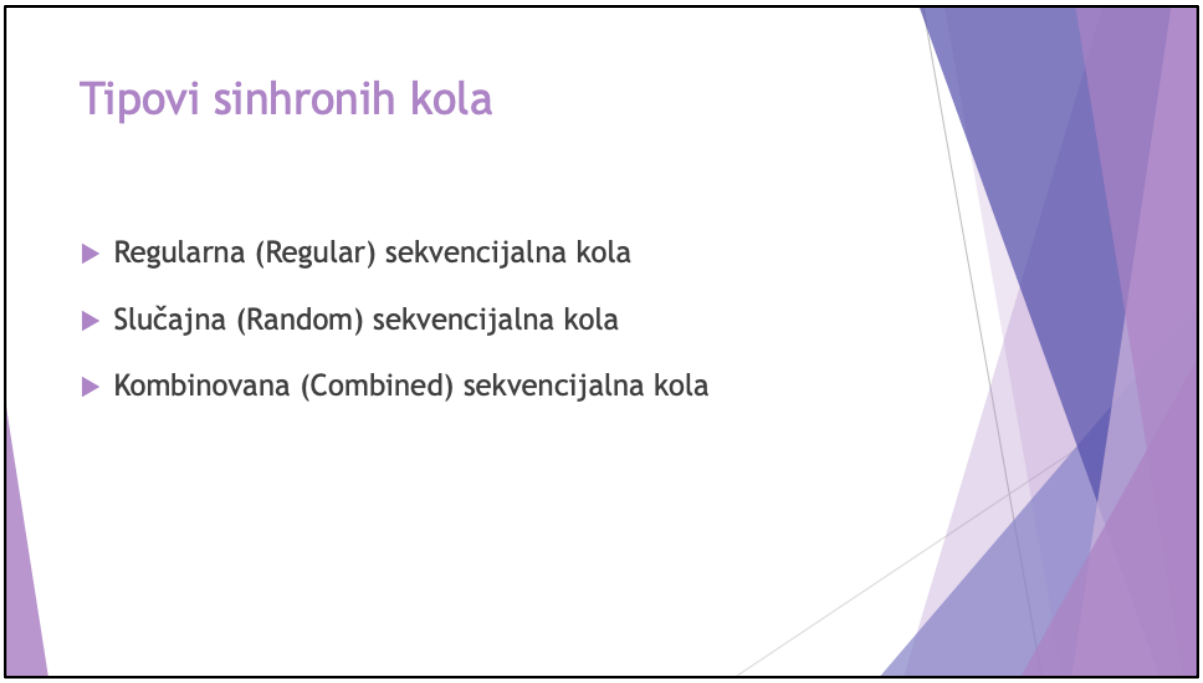


Na uzlaznu ivicu takt signala, vrijednlst state_next signala se odabira i prosljeđuje na q izlaz, i na taj način postaje nova vrijednost state_reg signala. Vrijednost se takođe smješta u FF i ostaje nepromijenjena do kraja periode takt impulsa. To predstavlja trenutno stanje sistema.

Na bazi state_reg signala i external input signala , next-state logic blok računa vrijednost state_next signala, dok output logic blok računa vrijednost output signala.

Na sljedeću uzlaznu ivicu takt signala, nova vrijednost state_next signala se odabira i state_reg signal je update-ovan. Nakon toga, proces se ponavlja.

Perioda takt signala mora biti dovoljna kako bi se ”uklopilo” propagaciono kašnjenje next-state logic, clock-to-q kašnjenje FF-a, kao i setup vrijeme FF-a.



Kod regularnih sekvencijalnih kola reprezentacija stanja i prelazi između stanja su jednostavni, regularni, kao što je slučaj kod brojača ili šift registra. Slično, next-state logic se može implementirati regularnim strukturalnim komponentama kao što su kolo za inkrementiranje i shifter.

Kod slučajnih sekvencijalnih kola prelazi između stanja su složeniji, bez posebne uslovljenosti stanja i njihove binarne reprezentacije. Next/state logic se mora izvoditi na osnovu scratch-a (slučajnom logikom). Ovaj tip kola je poznat kao Finite State Machine (FSM)

Kombinovana sekvencijalna kola se sastoje od regularnih sekvencijalnih kola i FSM-a. FSM se koristi kao kontrola regularnog sekvencijalnog kola. Ovaj tip kola se bazira na register transfer methodology i poznat je kao Finite State Machine with Data path (FSMD).

D latch

```
entity dlatch is
    port
        (
        c : in std_logic;
        d : in std_logic;
        q : out std_logic
        );
end dlatch;
architecture arch of dlatch is
begin
    process(c,d)
    begin
        if (c = '1') then
            q <= d;
        end if;
    end process;
end arch;
```

Positive-edge-triggered DFF

```
entity dff is
    port
        (
        clk : in std_logic;
        d : in std_logic;
        q : out std_logic
        );
end dff;
architecture arch of dff is
begin
    process(clk)
    begin
        if (clk'event and clk = '1') then
            q <= d;
        end if;
    end process;
end arch;
```

```
if rising_edge(clk) then
```

'event vraća **true** ako dođe do promjene u signalu na koji se odnosi. Uslovom **`if (clk'event and clk = '1')`** se ispituje da li je u pitanju rastuća ivica **`clk`** signala.

Negative-edge-triggered DFF

```
entity dff is
    port
        (
        clk : in std_logic;
        d : in std_logic;
        q : out std_logic
        );
end dff;
architecture arch of dff is
begin
    process(clk)
    begin
        if (clk'event and clk = '0') then
            q <= d;
        end if;
    end process;
end arch;
```

```
if falling_edge(clk) then
```

DFF sa asinhronim resetom

```
entity dffr is
    port
        (
        clk : in std_logic;
        reset : in std_logic;
        d : in std_logic;
        q : out std_logic
        );
end dffr;
architecture arch of dffr is
begin
    process(clk, reset)
    begin
        if (reset = '1') then
            q <= '0';
        elsif (clk'event and clk = '1') then
            q <= d;
        end if;
    end process;
end arch;
```

Asinhroni reset se koristi pri uključenju sistema.Uloga mu je inicijalizacija sistema, i nije poželjno koristit ga u toku rada sistema.

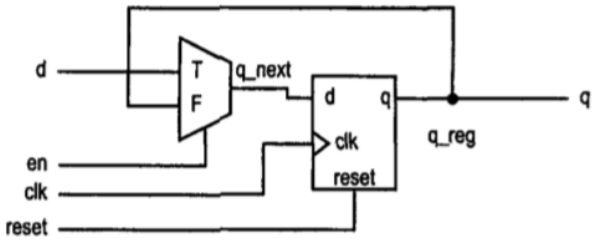
Register

```
entity reg8 is
    port
        (
        clk : in std_logic;
        reset : in std_logic;
        d : in std_logic_vector(7 downto 0);
        q : out std_logic(7 downto 0)
        );
end reg8;
architecture arch of reg8 is
begin
    process(clk, reset)
    begin
        if (reset = '1') then
            q <= (others => '0');
        elsif (clk'event and clk = '1') then
            q <= d;
        end if;
    end process;
end arch;
```

Registar je niz DFF-a koji koriste isti takt i reset signal.

DFF with enable

reset	clk	en	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↓	0	q
0	↓	1	d



DFF with enable

```
entity dff_en is
 port
     (
     clk, reset, en : in std_logic;
     d : in std_logic;
     q : out std_logic
     );
end dff_en;
architecture arch of dff_en is
     signal q_reg, q_next : std_logic;
begin
 process(clk, reset)
 begin
     if (reset = '1') then
         q <= '0';
     elsif (clk'event and clk = '1') then
         q_reg <= q_next;
     end if;
 end process;
 q_next <= d when en = '1' else
        q_reg;
 q <= q_reg;
end arch;
```

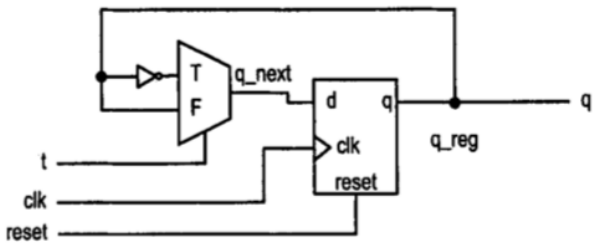
DFF

next-state logic

output logic

TFF

reset	clk	t	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↓	0	q
0	↓	1	q'



TFF

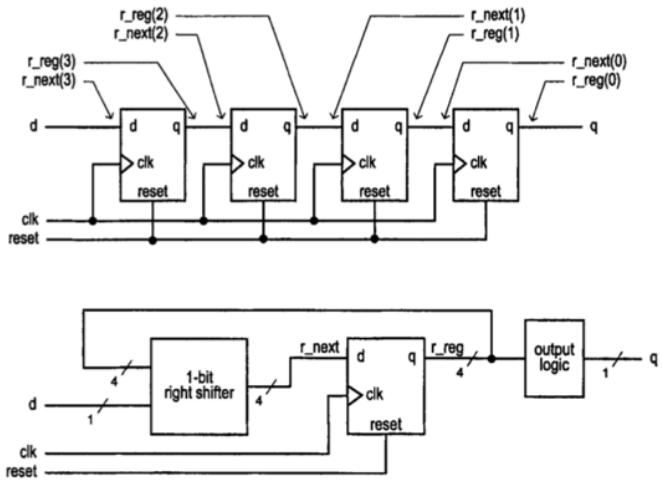
```
entity tff is
port
    (
    clk, reset : in std_logic;
    t : in std_logic;
    q : out std_logic
    );
end tff;
architecture arch of tff is
    signal q_reg, q_next : std_logic;
begin
process(clk, reset)
begin
    if (reset = '1') then
        q <= '0';
    elsif (clk'event and clk = '1') then
        q_reg <= q_next;
    end if;
end process;
q_next <= q_reg when t = '0' else
      not(q_reg);
q <= q_reg;
end arch;
```

DFF

next-state logic

output logic

Free-running shift-right register



Shift register pomjera svoj sadržaj lijevo ili desno za jedno mjesto u svakom ciklusu takt signala.

Free-running shift-right register

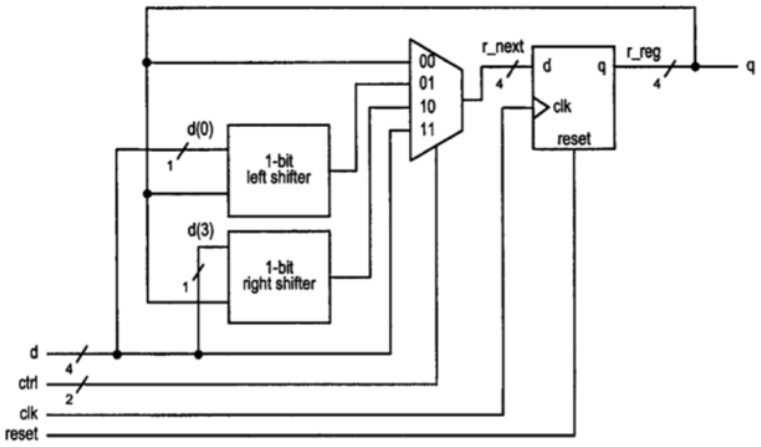
```
entity shift_right_register is
 port
     (
     clk, reset : in std_logic;
     d : in std_logic;
     q : out std_logic
     );
end shift_right_register ;
architecture arch of shift_right_register is
     signal r_reg, r_next : std_logic_vector(3 downto 0);
begin
 process(clk, reset)
 begin
     if (reset = '1') then
         r_reg <= (others => '0');
     elsif (clk'event and clk = '1') then
         r_reg <= r_next;
     end if;
 end process;
 r_next <= d & r_reg(3 downto 1);
     q <= r_reg(0);
end arch;
```

register

next-state logic

output logic

Universal shift register



Univerzalni **shift register** paralelno učitava podatak i izvršava pomjeranje u bilo kom smjeru. Operacije su sljedeće: **load, shift right, shift left** i **pause**. Kontrolnim signalom **ctrl** odabira se jedna od operacija.

Universal shift register

```
entity shift_register is
port
    (
    clk, reset : in std_logic;
    ctrl : in std_logic_vector(1 downto 0);
    d : in std_logic_vector(3 downto 0);
    q : out std_logic_vector(3 downto 0)
    );
end shift_register ;
architecture arch of shift_register is
    signal r_reg, r_next : std_logic_vector(3 downto 0);
begin
process(clk, reset)
begin
    if (reset = '1') then
        r_reg <= (others => '0');
    elsif (clk'event and clk = '1') then
        r_reg <= r_next;
    end if;
end process;
```

register

Universal shift register

next-state logic

```
with ctrl select
       r_next <=
           r_reg                        when "00", -- pause
           r_reg(2 downto 0) & d(0)     when "01", -- shift left
           d(3) & r_reg(3 downto 1)     when "10", -- shift right
           d                            when others; -- load
```

output logic

```
q <= r_reg;
```

```
end arch;
```

Arbitrary-sequence counter

```
entity arbi_seq_counter4 is
port
    (
    clk, reset : in std_logic;
    q : out std_logic_vector(2 downto 0)
    );
end arbi_seq_counter4 ;
architecture arch of arbi_seq_counter4 is
    signal r_reg, r_next : std_logic_vector(2 downto 0);
begin
process(clk, reset)
begin
    if (reset = '1') then
        r_reg <= (others => '0');
    elsif (clk'event and clk = '1') then
        r_reg <= r_next;
    end if;
end process;
```

register

Input pattern	Next pattern
000	011
011	110
110	101
101	111
111	000

Arbitrary-sequence counter

```
    r_next <= "011" when r_reg = "000" else
              "110" when r_reg = "011" else
              "101" when r_reg = "110" else
              "111" when r_reg = "101" else
              "000"; -- r_reg = "111"
```

next-state logic

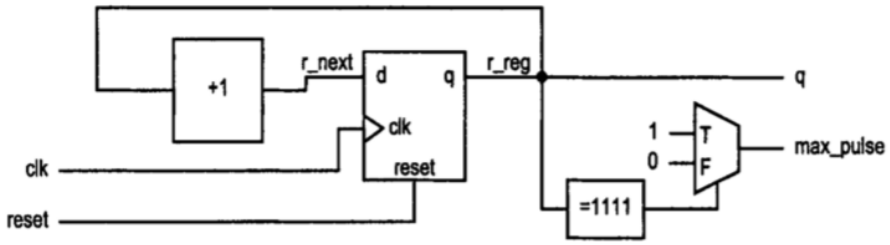
```
q <= r_reg;
```

output logic

```
end arch;
```

Input pattern	Next pattern
000	011
011	110
110	101
101	111
111	000

Free-running binary counter



Free-running binary counter

```
entity binary_counter4_pulse is
 port
     (
     clk, reset : in std_logic;
     max_pulse : out std_logic;
     q : out std_logic_vector(3 downto 0)
     );
end binary_counter4_pulse;
architecture arch of binary_counter4_pulse is
     signal r_reg, r_next : unsigned(3 downto 0);
begin
     process(clk, reset)
     begin
         if (reset = '1') then
             r_reg <= (others => '0');
         elsif (clk'event and clk = '1') then
             r_reg <= r_next;
         end if;
     end process;
     r_next <= r_reg + 1;
     q <= std_logic_vector(r_reg);
     max_pulse <= '1' when r_reg = "1111" else
                  '0';
end arch;
```

register

next-state logic

output logic

Featured binary counter

```
entity binary_counter4_feature is
port
    (
    clk, reset : in std_logic;
    syn_clr, en, load : in std_logic;
    d : in std_logic_vector(3 downto 0);
    q : out std_logic_vector(3 downto 0)
    );
end binary_counter4_feature;
architecture arch of binary_counter4_feature is
    signal r_reg, r_next : unsigned(3 downto 0);
begin
process(clk, reset)
begin
    if (reset = '1') then
        r_reg <= (others => '0');
    elsif (clk'event and clk = '1') then
        r_reg <= r_next;
    end if;
end process;
```

register

operation	syn_clr	load	en	q
Sync clear	1	-	-	00 ... 00
Parallel load	0	1	-	d
count	0	0	1	q+1
pause	0	0	0	q

Featured binary counter

```
    r_next <= (others => '0') when sync_clr = '1' else
              unsigned(d)     when load = '1' else
              r_reg + 1       when en = '1' else
              r_reg;
```

next-state logic

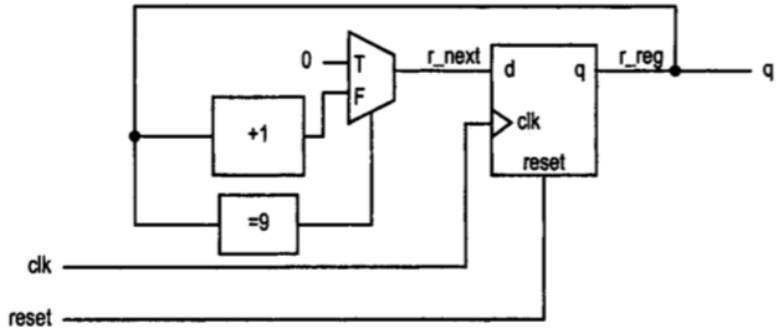
```
q <= std_logic_vector(r_reg);
```

output logic

```
end arch;
```

operation	syn_clr	load	en	q
Sync clear	1	-	-	00 ... 00
Parallel load	0	1	-	d
count	0	0	1	q+1
pause	0	0	0	q

Decade counter



Decade counter

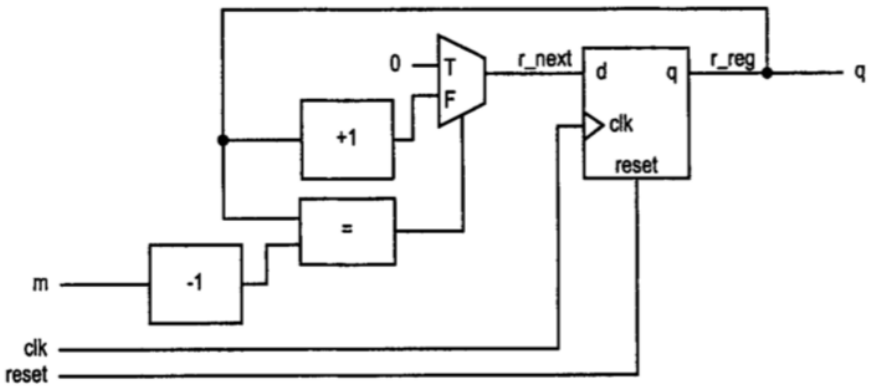
```
entity mod10_counter is
port
    (
    clk, reset : in std_logic;
    q : out std_logic_vector(3 downto 0)
    );
end mod10_counter;
architecture arch of mod10_counter is
    constant TEN : integer := 10;
    signal r_reg, r_next : unsigned(3 downto 0);
begin
process(clk, reset)
begin
    if (reset = '1') then
        r_reg <= (others => '0');
    elsif (clk'event and clk = '1') then
        r_reg <= r_next;
    end if;
end process;
r_next <= (others => '0') when r_reg = (TEN-1) else
            r_reg + 1;
q <= std_logic_vector(r_reg);
end arch;
```

register

next-state logic

output logic

Programmable mod-*m* counter



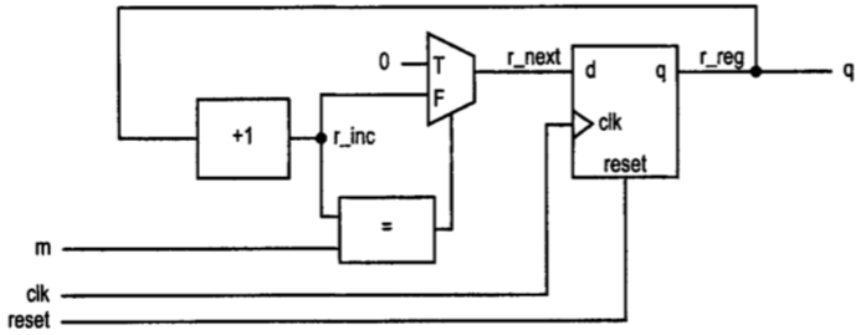
Blok dijagram inicijalnog dizajna

Programmable mod-*m* counter

```
entity prog_counter is
port
    (
    clk, reset : in std_logic;
    m : in std_logic_vector(3 downto 0);
    q : out std_logic_vector(3 downto 0)
    );
end prog_counter;
architecture arch of prog_counter is
    signal r_reg, r_next : unsigned(3 downto 0);
begin
process(clk, reset)
begin                                          register
    if (reset = '1') then
        r_reg <= (others => '0');
    elsif (clk'event and clk = '1') then
        r_reg <= r_next;
    end if;
end process;
r_next <= (others => '0') when r_reg = (unsigned(m) - 1) else
            r_reg + 1;                         next-state logic
q <= std_logic_vector(r_reg);                  output logic
end arch;
```

next-state logic sadrži kolo za inkrementiranje i kolo za dekrementiranje. Izraz `r_reg = (unsigned(m) - 1)` se može zapisati i kao `(r_reg + 1) = unsigned(m)`

Programmable mod-*m* counter



Blok dijagram efikasnijeg dizajna

Programmable mod-*m* counter

```
entity prog_counter is
port
    (
    clk, reset : in std_logic;
    m : in std_logic_vector(3 downto 0);
    q : out std_logic_vector(3 downto 0)
    );
end prog_counter;
architecture arch of prog_counter is
    signal r_reg, r_next, r_inc : unsigned(3 downto 0);
begin
process(clk, reset)
begin
    if (reset = '1') then
        r_reg <= (others => '0');
    elsif (clk'event and clk = '1') then
        r_reg <= r_next;
    end if;
end process;
r_inc <= r_reg + 1;
r_next <= (others => '0') when r_inc = unsigned(m) else
          r_inc;
q <= std_logic_vector(r_reg);
end arch;
```

register

next-state logic

output logic

next-state logic sadrži kolo za inkrementiranje i kolo za dekrementiranje. Izraz `r_reg = (unsigned(m) - 1)` se može zapisati i kao `(r_reg + 1) = unsigned(m)`

Programmable mod-*m* counter, one-segment design

```
architecture not_work_one_arch of prog_counter is
    signal r_reg : unsigned(3 downto 0);
begin
 process(clk, reset)
 begin
    if (reset = '1') then
        r_reg <= (others => '0');
    elsif (clk'event and clk = '1') then
        r_reg <= r_reg + 1;
        if (r_reg = unsigned (m)) then
            r_reg <= (others => '0');
        end if;
    end if;
 end process;
q <= std_logic_vector(r_reg);
end not_work_one_arch;
```

Ovaj kod ne funkcioniše onako kako na prvi pogled izgleda. Naime, signalima se vrijednost dodjeljuje na kraju procesa. Dakle, **r_reg** će biti inkrementirano na kraju procesa, što znači da će izraz **`r_reg = unsigned (m)`** ispitivati jednakost **`unsigned(m)`** sa straom vrijednošću **`r_reg`**. Brojač će da broji za jedan više.

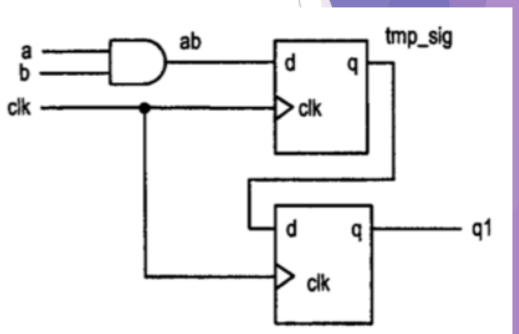
Programmable mod-*m* counter, one-segment design

```
architecture work_one_arch of prog_counter is
    signal r_reg, r_inc : unsigned(3 downto 0);
begin
 process(clk, reset)
 begin
    if (reset = '1') then
        r_reg <= (others => '0');
    elsif (clk'event and clk = '1') then
        if (r_inc = unsigned (m)) then
             r_reg <= (others => '0');
        else
            r_reg <= r_inc;
        end if;
    end if;
 end process;
r_inc <= r_reg + 1;
q <= std_logic_vector(r_reg);
end work_one_arch;
```

Potrebno je inkrementiranje obaviti van procesa, kako bi se izvršilo paralelno sa procesom.

Upotreba varijabli u opisu sekvencijalnih kola

```
entity variable_ff_demo is
    port
        (
        a, b, clk : in std_logic;
        q1, q2, q3 : out std_logic
        );
end variable_ff_demo;
architecture arch of variable_ff_demo is
    signal tmp_sig1 : std_logic;
begin
-- attempt 1 --
    process(clk)
    begin
        if(clk'event and clk = '1') then
            tmp_sig1 <= a and b;
            q1 <= tmp_sig1;
        end if;
    end process;
```



Osnovna uloga varijabli jeste dobijanje međurezultata unutar **clk'event and clk='1'** grane bez uvođenja neželjenog registra. Slijedi primjer.

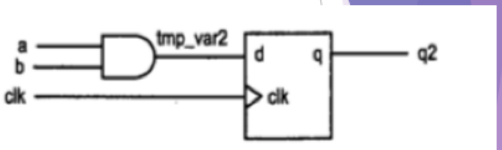
U prvom pokušaju, namjera je bila iskoristiti signal **tmp_sig1** za smještanje privremenog reultata. Međutim, kako je signal **tmp_sig1** unutar **clk'event and clk='1'** grane, uveden je **DFF**. Izrazi:

```
tmp_sig1 <= a and b;
q1 <= tmp_sig1;
```

se interpretiraju na sljedeći način. Na uzlaznu ivicu **clk** signala, vrijednost **a and b** će se odabrati i smjestiti u **FF** po nazivu **tmp_sig1**, dok će stara vrijednsot (ne trenutna vrijednost **a and b**) iz **tmp_sig1** signala biti smještena u **FF** po nazivu **q1**. Dijagram je prikazan na slici. Vrijednost **a and b** je vremenski pomjerena zbog uvođenja neželjenog memorijskog elementa (bafera) i sistem se ne ponaša kako je planirano. Kako je u pitanju **signal assignment statement**, neće biti nikakve razlike ukoliko se redosljed izraza izmijeni.

Upotreba varijabli u opisu sekvencijalnih kola

```
-- attempt 2 --
process(clk)
    variable tmp_var2 : std_logic;
begin
    if(clk'event and clk = '1') then
        tmp_var2 := a and b;
        q2 <= tmp_var2;
    end if;
end process;
-- attempt 3 --
process(clk)
    variable tmp_var3 : std_logic;
begin
    if(clk'event and clk = '1') then
        q3 <= tmp_var3;
        tmp_var3 := a and b;
    end if;
end process;
end arch;
```



U drugom pokušaju, koristi se varijabla tmp_var2 za smještanje privremenog rezultata:

```
tmp_var2 := a and b;
q2 <= tmp_var2;
```

Varijabli tmp_var2 je najprije dodijeljena vrijednost, a potom je korišćena za dodjelu vrijednosti signalu. Dakle, nisu uvođeni dodatni memorijski elementi i kolo radi kako je planirano. Blok dijagram je prikazan na slici.

U trećem pokušaju, varijabla **tmp_var3** je korišćena za smještanje privremenog rezultata, međutim redosljed izraza je izmijenjen:

```
q3 <= tmp_var3;
tmp_var3 := a and b;
```

Varijabla je korišćena prije nego što joj je dodijeljena vrijednsot. Po VHDL definiciji, koristiće se vrijednost iz prethodnog izvršavanja procesa. FF se uvodi za

smještanje te prethodne vrijednosti. Slijedi da je kolo opisano u trećem pokušaju isto po funkcionalnosti kao i kolo iz prvog pokušaja.

Programmable mod-*m* counter, one-segment design

```
architecture variable_arch of prog_counter is
    signal r_reg : unsigned(3 downto 0);
begin
 process(clk, reset)
    variable q_tmp : unsigned(3 downto 0);
 begin
    if (reset = '1') then
        r_reg <= (others => '0');
    elsif (clk'event and clk = '1') then
        q_tmp := r_reg + 1;
        if (q_tmp = unsigned (m)) then
             r_reg <= (others => '0');
        else
            r_reg <= q_tmp;
        end if;
    end if;
 end process;
q <= std_logic_vector(r_reg);
end variable_arch;
```

Problem do koga se došlo kod primjera one-segment programmable mod-m counter se može riješiti upotrebom varijable **q_tmp**. Za razliku od signala, varijabla preuzima vrijednost trenutno i sistem funkcioni[e kako je planirano.

Preporuke za sintezu

- Svi registri u sistemu treba da budu sinhronizovani zajedničkim globalnim takt signalom.
- Izolovati memorijske komponente od VHDL opisa i kodirati ih u okviru zasebnog segmenta. One-coding stil se ne preporučuje.
- Asinhroni reset, ukoliko se koristi, treba da služi samo za inicijalizaciju sistema. Ne treba ga koristiti za brisanje sadržaja registara u toku regularnog rada sistema.
- Varijable ne treba koristiti za uvođenje memorijskog elementa.